

ECOLE NATIONALE SUPÉRIEURE DE L'AÉRONAUTIQUE
ET DE L'ESPACE

BE IN201 : Rapport Final
Programme de lancer de rayon en java

Emmanuel Branlard - Dimitri Sluys

Le 3 Mars 2008

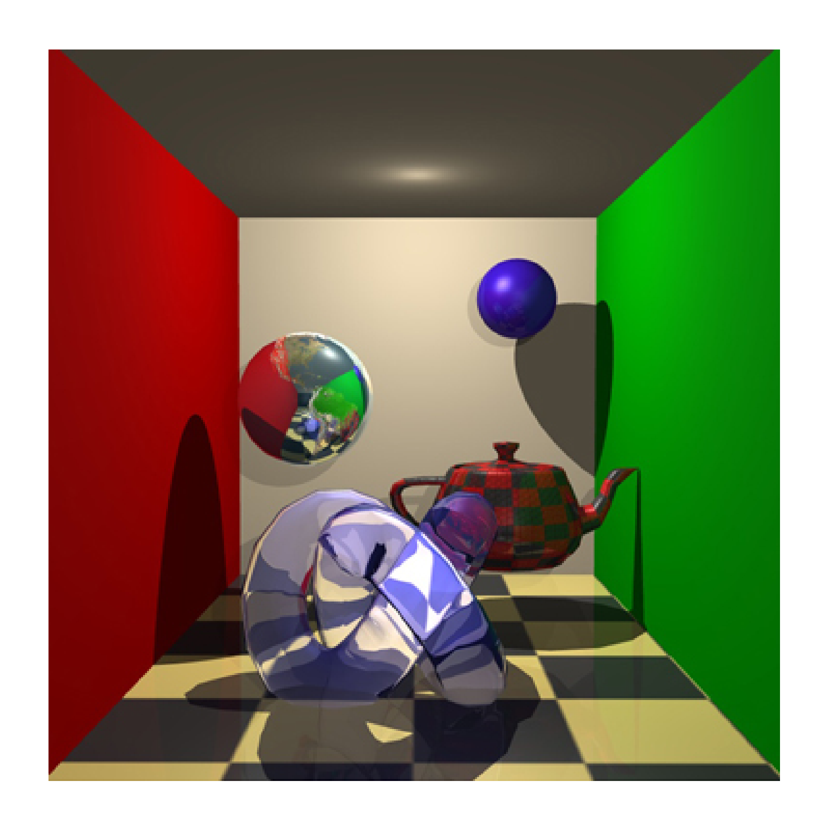


Table des matières

Introduction

Ce rapport établit un bilan du travail effectué sur le projet du bureau d'étude de programmation JAVA. Il se compose de trois parties. Les deux premières sont la reproduction identique des deux précédents rapports, cela afin d'éviter au lecteur de devoir relire intégralement des choses qu'il a déja lues cherchant les fameuses sept différences. Toutefois, le lecteur pourra remarquer que les classes de la deuxième partie ont changé de couleur par rapport au deuxième rapport. En effet, nous avons changé de style de diagramme UML sous Eclipse, afin de mieux mettre en valeur le type des paramètres. Il pourrait donc être interressant pour le lecteur de jetter un oeil sur ces nouveaux diagrammes car ils sont plus précis et plus facilement compréhensibles grace à la présence des types. Les classes n'ont pas ou très peu évolué entre temps, les différences entre les deux versions devraient être imperceptibles. La troisième partie du rapport est quant à elle "nouvelle" et présente le programme final.

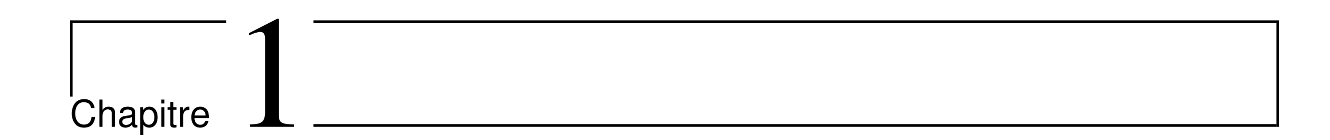
Analyse des besoins et préparation à la validité : Cette partie présente la phase d'analyse du projet du bureau d'étude. Nous simulons ici une situation réelle où nous prenons la place du concepteur qui doit présenter le résultat de ces analyses auprès de son client. Il apparaît nécessaire de définir les besoins du client. Nous ferons ceci à travers l'étude d'un cas d'utilisation qui nous permettra de faire ressortir les exigences de l'utilisateur. Dans un second temps, avant de livrer notre programme il faudra effectuer une série de tests. Notre troisième partie décrira succintement l'interface du programme.

Solution de conception répondant aux besoins : Cette partie présente la phase de conception du projet du bureau d'étude. Afin de dégager l'esprit et le fonctionnement de notre programme nous présenterons les classes et packages les plus importants. Nous tâcherons de mettre en valeur les interactions entre les différentes classes à l'aide de diagrammes de séquence. Dans un dernier temps nous présenterons des methodes pour tester certaines classes, puis un plan de developpement définissant la répartition du travail au sein de notre binôme.

Le programme final et ses évolutions : Afin de clore notre travail sur ce projet, nous avons tenu à ce que le lecteur puisse comprendre le fonctionnement de notre programme. C'est pourquoi, après avoir effectué dans un premier temps un manuel d'utilisation, nous présenterons quelques point clés du programme, sous forme l'algorithmes et de schémas. Enfin, nous évoquerons les changements qui ont eu lieu entre la phase de conception et l'implémentation.

Première partie

Analyse des besoins et préparation à la validation



Le rôle du logiciel pour son utilisateur

1.1 Définition d'un cas d'utilisation

L'utilisateur fournit un fichier décrivant une scène. Celle-ci reconstitue par exemple la position des planètes du système solaire, centrées au niveau du soleil, à un instant t fixé. Ce fichier est défini de manière ordonnée en présentant les différents paramètres du problème à résoudre (positions des objets et des sources, tailles, intensités, caractéristiques matériaux et couleurs). Nous ne nous prononçons pas encore sur la façon de construire ce fichier, manuelle ou informatisée. Ces fichiers de description de scène seront constitués de sphères et plans infinis ou de facettes. L'utilisateur choisit alors de positionner l'observateur de la scène ainsi que la taille de l'écran, afin d'avoir une certaine vue de notre galaxie.

Nous laisserons également libre à celui-ci de choisir la profondeur de lancer de rayon (grossièrement cela correspond aux différents ordres de réflexion/réfraction sur les objets de la scène). Notre logiciel produira ainsi l'image perçue par l'observateur et l'affichera à l'écran.



Fig. 1.1 – Diagramme des étapes d'un cas d'utilisation

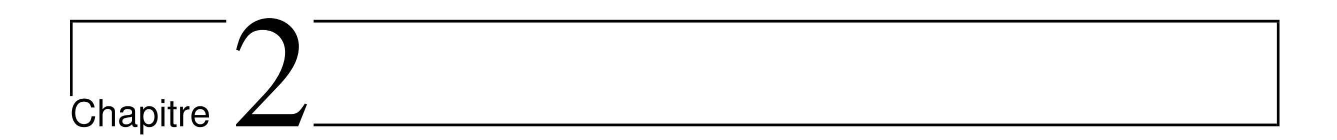
1.2 Les besoins de l'utilisateur

Son besoin est la fonction principale du logiciel, à savoir, de reconstituer sur un écran ce que voit un observateur d'une scène constituée d'objets géométriques et de sources de lumière colorée. Il veut pouvoir modifier les paramètres de la scène (position et champ de vision de l'observateur, ce que nous appelons ici point de vue et taille de l'écran). Dans l'idéal, ce type de logiciel a une application au domaine des films en images de synthèse, et il s'insèrerait donc dans un processus d'image par image permettant de réaliser une animation. Nous n'irons certainement pas jusque-là mais il est évident que l'utilisateur aura besoin d'exporter l'image produite.

En ce qui concerne l'extensibilité du logiciel, c'est-à-dire créer de nouveaux objets et changer le modèle de lumière, cela ne relève pas du domaine de compétence de l'utilisateur mais de celui du concepteur futur. Il ne s'agit donc pas d'une fonctionnalité requise et accessible par l'utilisateur actuel.

1.3 Ce dont l'utilisateur a besoin

Il sera nécéssaire à toute personne voulant utiliser le logiciel manuellement de connaître la syntaxe des fichiers source décrivant la scène. Ce dernier aura besoin d'un manuel d'utilisation du logiciel afin d'en maîtriser ses fonctionnalités et de comprendre l'effet de chaque réglages (taille de l'écran, profondeur du lancer de rayon).



La validation du programme

2.1 Est-ce que notre programme fonctionne correctement

Nous devrons dans un premier temps vérifier que les différents modules implémentent correctement les exigences du client. Il s'agit ici d'effectuer lors de la conception des test en JUnit afin de voir si chaque composant du programme réalise bien son cahier des charges. De plus, il faudra, une fois le programme terminé, vérifier que celui renvoie bien une image conforme à une situation réelle. Les couleurs, les ombres, la transparence, les superpositions doivent être transcrites fidèlement à la scène.

2.2 Plan de tests

Nous proposons d'effectuer des scènes "élémentaires" permettant de tester les points principaux du programme :

- Objet seul dans l'espace
- Variation du point de vue et de la taille de l'écran
- Les ombres sont bien placées
- Les objets se cachent entre eux
- La réfraction et la réflexion sur les objets

Bien sûr cette liste n'est pas exhaustive, et il se peut que celle-ci soit complétée ultérieurement à mesure que nous avancerons dans le projet. De plus, ce n'est pas parce qu'un cas de figure semble refléter correctement la réalité qu'il n'existe pas de problème.

2.3 Est-ce que notre programme répond aux attentes des cas d'utilisation par le client

Durant une seconde phase, nous reproduirons des cas d'utilisation typiques que le client peut être ammené à effectuer. Cela peut se faire en faisant tester directement le client, ou des personnes externes à la conception du programme. Un cas d'utilisation typique a été décrit dans la section 4.1.

2.4 Est-ce que notre programme est stable et performant

Afin de terminer cette phase de tests, il apparaît nécessaire de vérifier la conformité de la solution par rapport à ses exigences de performance. Ainsi nous testerons la robustesse et la stabilité du programme. Nous nous souviendrons du problème de la place mémoire qui était une des principales préoccupations de notre BE de l'année dernière. De plus, l'utilisateur attend certainement une réponse dans un délai acceptable, ainsi le problème prend une dimension temporelle qu'on sera tenu de prendre en compte dans cette phase.





Chapitre 3

La future interface graphique

3.1 L'interface en image

Fig. 3.1 – Schéma de l'interface graphique

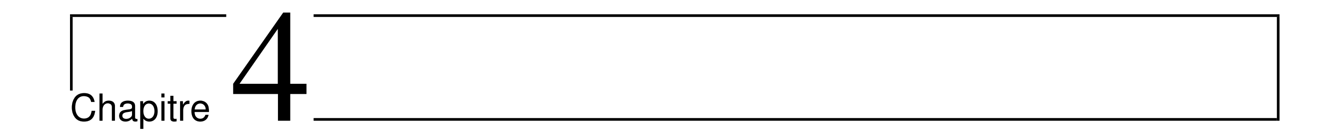
3.2 Descriptif de l'interface

L'interface est composé de trois parties. Une partie supérieure où l'on affiche le nom du fichier chargé. Une partie gauche contenant le menu d'utilisation aux fonctionnalités suivantes :

chargement d'un fichier, modification et sauvegarde des paramètres de vue, affichage de l'image, exportation de celle-ci. Enfin, une zone principale est réservée à l'affichage de l'image.

Deuxième partie

Solution de conception répondant aux besoins



Description globale des classes

4.1 Diagramme général du programme

4.1.1 Les relations d'associations



L'analyse du projet nous a amené à dégager deux aspects du programme : l'interface graphique d'une part, et ce que nous appellerons le theatre d'autre part. Nous avons créé une classe $Main$, dans un package Main, qui sera la seule classe disposant d'un main de notre programme (classes de test exclues). Celle-ci créera alors une Fenetre correspondant à l'interface graphique et un théatre correpondant aux données du problème.

Voici alors le diagramme général simplifié de l'ensemble des classes nécessaires à ce jour pour le bon fonctionnmenet du logiciel.

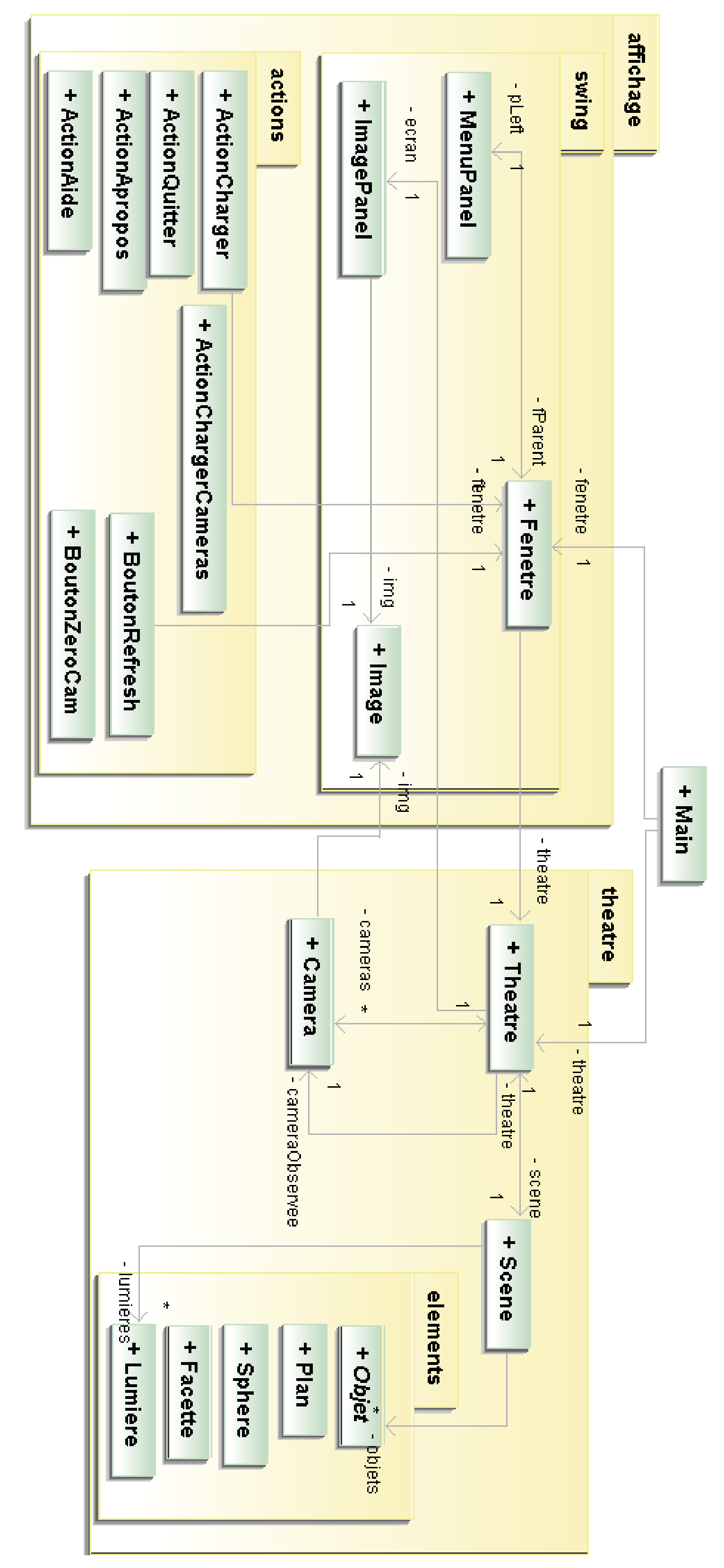


Fig. 4.1 – Diagramme des classes

Nous avons envisagé d'autres packages, que nous n'avons pas inclu dans le diagramme ci-dessus car souvent solicités.

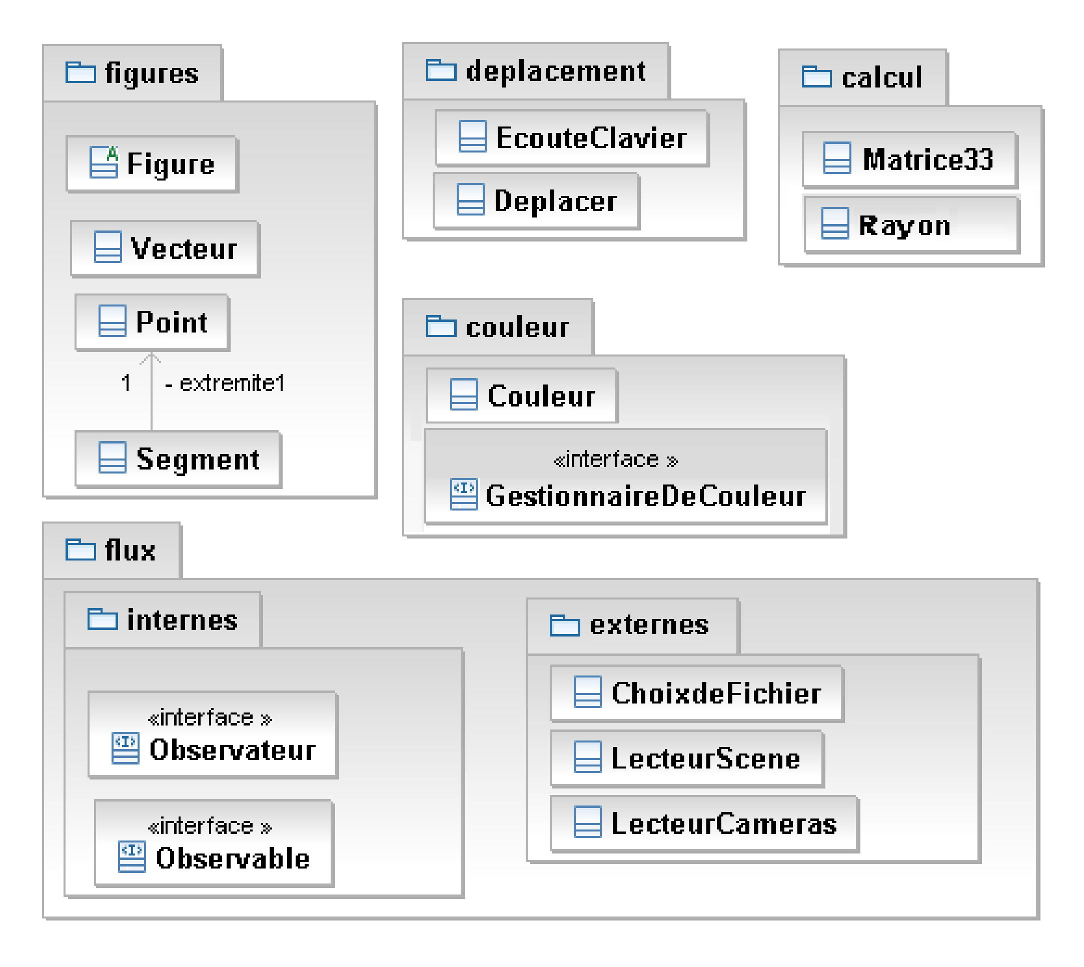


Fig. 4.2 – Diagramme des packages fréquemment utilisés

4.1.2 Le problème d'héritage...

Nous n'avons pas fait figurer sur le diagramme précédent les relations d'héritage entre les différentes classes. Voici donc, sous forme textuelle, les relations d'héritage :

- *Plan*, *Sphere*, et *Facette* héritent de la classe abstraite *Objet*
- *Image* hérite de *BufferedImage*
- *Vecteur*, hérite de la classe abstraite *Figure*
- *Point*, hérite de la classe abstraite *Vecteur*
- *PhongBlinn* réalise l'interface de *GestionnaireDeCouleur*
- *Fenetre* hérite de *JFrame*
- *ImagePanel* hérite de *JPanel*
- *ActionCharger*, *ActionAide*, *ActionAPropos*, *ActionQuitter* héritent de *AbstractAction*
- *MenuPanel* hérite de *JPanel* et réalise l'interface de *ActionListener* (pour écouter les actions de l'utilisateur (choix de caméra et bouton refresh)
- *EcouteClavier* réalise l'interface *KeyListener*

4.2 Le Théâtre

Dans la suite, nous avons parfois choisi de mettre sur nos diagrammes UML les attributs, car ceux-ci permettent de bien comprendre ce qui constitue nos classes.

4.2.1 La classe theatre

```
+ Theatre
- cameras: Vector<Camera>
- cameraObservee: Camera
- scene: Scene
- ecran: ImagePanel
FichierCameras: String
+ Theatre()
+ chargeDefaultCamera()
+ setEcran(IP: ImagePanel)
+ getCameraObservee(): Camera
+ setCameraObservee(cam: Camera)
+ getEcran(): ImagePanel
+ getCameras(): Vector<Camera>
+ getScene(): Scene
+ toString(): String
```

Après analyse, nous avons choisi de créer un objet particulier : le théâtre . Un théâtre est composé d'une scène, de caméras, et d'un lien vers un écran. Nous avons remplacer la notion d'observateur par celle de caméra (ou appareil photo). Nous ne nous restreindrons pas à une seule caméra, cela permettant d'aller d'un point de vue à un autre facilement, comme au théâtre ou au cinéma. Selon nous, la scène est un ensemble d'objets, elle est fixe, et correspond à un unique fichier de configuration. Au sein du théâtre, on choisit une caméra, la "caméraObservée", que l'on communique à l'écran. Ce dernier, se charge d'afficher l'image vue par cette caméra au sein de l'interface graphique. Le set de caméras initial est situé dans un fichier que l'on charge au démarrage.

4.2.2 La classe scene

```
+ Scene
- theatre: Theatre
- file: String
- objets: Vector<Objet>
- lumieres: Vector<Lumiere>
+ erreur: int
+ Scene(theatre_: Theatre)
+ setFile(ch: String)
+ plusPetitObjetContenant(P: Point): Obje
+ getFile(): String
+ getObjets(): Vector<Objet>
+ getLumieres(): Vector<Lumiere>
+ toString(): String
```

Une scène contient des lumières et des objets. Elle charge ces données via un fichier de configuration, celui-ci devant respecter la norme donnée dans l'énoncé. Lorsque l'utilisateur désire charger une scène, il choisit un nom de fichier. En appelant alors la fonction *setFile*(), la classe charge ce fichier et se met à jour. La scène sait à quel théâtre elle appartient via la variable *theatre*. Cela pourra par exemple lui permettre de réafficher l'image de la caméra observée si un nouveau fichier de configuration est ouvert.

4.2.3 La classe objet

+ *Objet*

coeffref: double
coefftrans: double
couleur: Couleur
indice: double
brillance: double
reflectance: double

+ Objet(coeffref_: double, coefftrans_: double, couleur_: Couleur, indice_: double, reflectance: double, brillance: int)
+ *estIntersecte(R: Rayon): Point*
+ *intersectionSuivante(R: Rayon): Point*
+ *estUnContenant(): boolean*
+ *toString(): String*
+ *estDansObjet(P: Point): boolean*
+ *rentreDansObjet(R: Rayon): boolean*
+ getCouleur(): Couleur
+ getCoeffref(): double
+ getCoefftrans(): double
+ getReflectance(): double
+ getBrillance(): double
+ getIndice(): double
+ *getNormale(P: Vecteur): Vecteur*

Fig. 4.3 – Diagramme UML de la classe Objet

Afin de permettre l'extension du programme à des objets autres que des sphères, des plans ou des facettes, nous avons créé une classe abstraite : la classe *Objet*. Elle contient les prototypes abstraits de chacune des méthodes de base que doivent contenir tout objet (Plan sphère, ou objet à facette). Les attribus n'ont pas été représentés ici. Les classes *Plan* et *Sphere* et *Facette* hériteront de la classe *Objet*.

4.2.4 La classe lumière

+ Lumiere

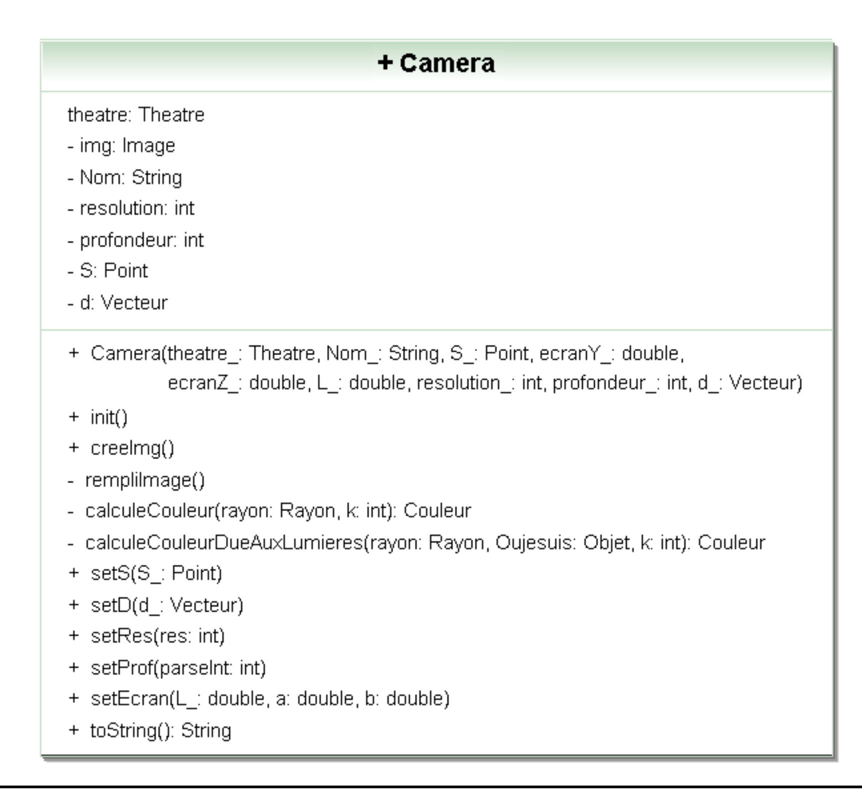
couleur: Couleur
S: Point
intensite: int

+ Lumiere(x_: double, y_: double, z_: double, couleur_: Couleur, intensite_: int)
+ getCouleur(): Couleur
+ getIntensite(): int
+ getS(): Point
+ toString(): String

Fig. 4.4 – Diagramme de la classe Lumiere

La classe *Lumiere* permet de représenter une Lumière sur la scène. On lui attribue une intensité qui est un nombre entier, correspondant au carré de la distance qu'un rayon lumineux peut parcourir. En effet le rapport intensité sur eclairement en un point est inversement proportionnel à la distance à la source au carré.

4.2.5 La classe camera



Comme nous l'avons vu plus haut, un theatre comprend différentes caméras. Une caméra, contient principalement l'image qu'elle peut voir. Ceci est fonction de sa position, de la direction dans laquelle elle regarde, mais également, de la taille et de la position de l'écran (ou fenêtre) correspondant à son champ de vision. Pour l'heure, c'est dans cette classe que nous effectuerons le calcul de la couleur d'un point.

4.2.6 Deux classes utiles : la classe Vecteur et la classe Rayon

+ Vecteur

- x: double
- y: double
- z: double

+ Vecteur(x_: double, y_: double, z_: double)
+ Vecteur(x_: double, y_: double, z_: double, couleur_: Color)
+ getX(): double
+ getY(): double
+ getZ(): double
+ setX(x_: double)
+ setY(y_: double)
+ setZ(z_: double)
+ translater(dx: double, dy: double, dz: double)
+ afficher()
+ toString(): String
+ distance(p: Vecteur): double
+ norme(): double
+ ps(p: Vecteur): double
+ det(p: Vecteur, q: Vecteur): double
+ pv(p: Vecteur): Vecteur
+ dir(p: Vecteur): Vecteur
+ moins(p: Vecteur): Vecteur
+ plus(p: Vecteur): Vecteur
+ fois(a: double): Vecteur
+ unitaire(): Vecteur
+ cosangle(p: Vecteur): double

Pour définir l'orientation du regard sur la scène, nous utilisons un vecteur. Nous nous armons également d'une multitude d'opérations élémentaires pour pouvoir modifier ou qualifier ce fameux vecteur. Ce vecteur sera primordial pour le jeu des lumières et le calcul de trajectoires de rayons.

```
+ Rayon
-----------------------------------------------------------
direction: Vecteur
Origine: Point
Intersection: Point
objetIntersecte: Objet
distanceParcourue: double
reflechi: Rayon
refracte: Rayon
rayon_precedent: Rayon
Normale: Vecteur
milieu: double
-----------------------------------------------------------
+ Rayon(Origine: Point, direction: Vecteur, n: double)
+ calculObjetIntersecte(objets: Vector<Objet>): Objet
+ getOrigine(): Point
+ getDirection(): Vecteur
+ getIntersection(): Point
+ setIntersection(I: Point)
+ setRayonPrecedent(R: Rayon)
+ getObjetIntersecte(): Objet
+ setObjetIntersecte(O: Objet)
+ getMilieu(): double
+ getDistanceParcourue(): double
+ getReflechi(): Rayon
+ getRefracte(): Rayon
+ getNormale(): Vecteur
+ toString(): String
```

De manière évidente, un rayon sera constitué d'un $Point$ Origine et d'un $Vecteur$ direction unitaire. En appelant la méthode $calculeObjetPlusProche()$ on déterminera alors, l'objet le plus proche qui est intersecté par ce rayon. Nous connaitrons alors, si intersection il y a : la distance entre ce rayon et l'objet, le point d'intersection, ainsi que les deux rayons réfléchi et réfracté produits.

4.3 Interface graphique

L'autre partie importante de la conception est l'inteface graphique. L'utilisateur doit pouvoir manier aisément le logiciel. Pour cela, nous doterons notre programme d'un certain nombre de commandes lui permettant de modifier la position des caméras (X,Y et Z observateur), de l'écran, et ce grâce à un bouton d'actualisation. Ceci est plus pratique que d'avoir à relancer le programme à chaque fois. Après reflexion, nous avons choisi d'ajouter à l'interface graphique un $JCombobox$ pour choisir la caméra que l'on souhaite observer. Ainsi, on pourra facilement passer d'une vue selon l'axe x, à une vue de trois-quart par exemple. Nous envisageons par ailleurs de permettre à l'utilisateur de déplacer la position de la caméra en cours, via un $KeyListener$, la classe $EcouteClavier$.

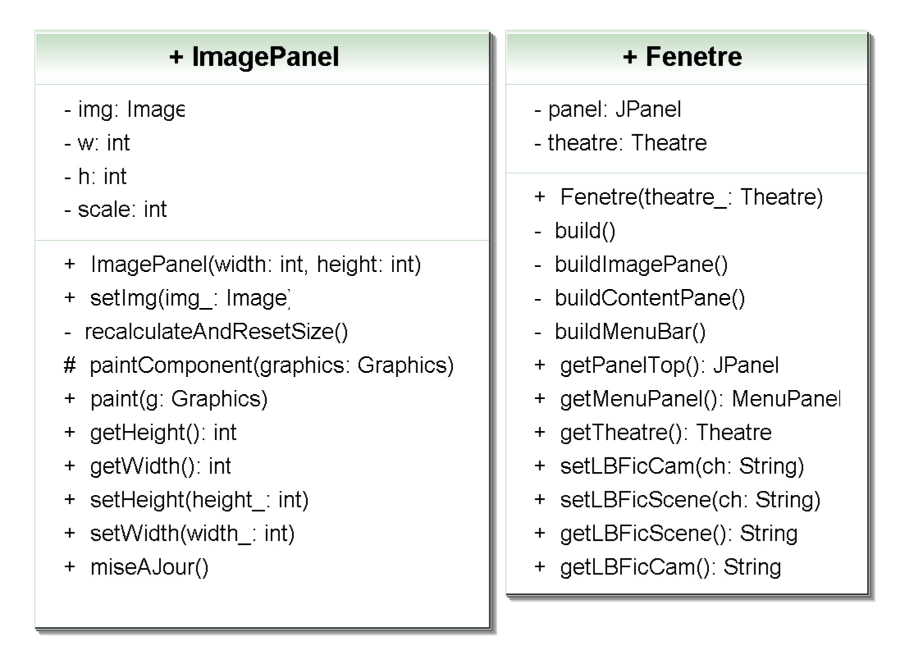


Fig. 4.5 – Diagrammes UML des classes Fenetre et ImagePanel

4.4 La gestion des couleurs

4.4.1 La modélisation des couleurs

Nous avions dans un premier temps choisi d'utiliser des couleurs exprimées en RGB, mais représentées sous la forme d'*int*. En effet, la classe *Color* du package *java.awt* permet de retourner via la méthode *getRGB*() un entier représentatif de la couleur. La position des bits dans cet entier permet de se référer soit au Rouge, soit au Vert, soit au Bleu. C'est pourquoi nos fonctions *getCouleur*() ci-dessus retournent toujours des types int, et non des tableaux de taille 3. Toutefois, nous avons reçu par la suite un mail nous indiquant d'utiliser des doubles entre 0 et 1. Chose que nous utiliserons lors du developpement. Toutefois, compte tenu de l'arrivée tardive de cette indication, nous n'avons pas eu le temps de modifier nos diagrammes UML. Suite au mail nous avons créé une classe *Couleur*, qui représente une couleur sous forme de trois doubles. De plus, afin de permettre une extension de la modélisation des couleurs, nous créons une classe *GestionnaireDeCouleur*, qui pourra ajouter des couleurs et calculer les différentes composantes de celle-ci.

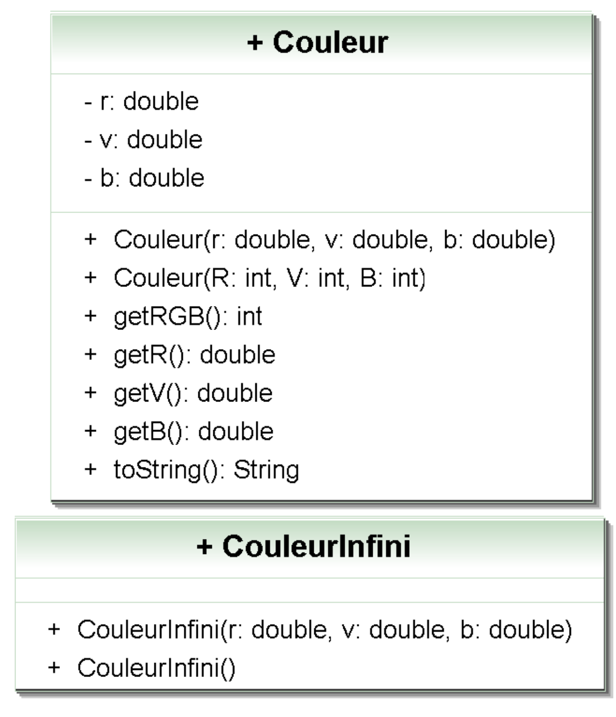


Fig. 4.6 – Diagrammes UML des classes couleur et couleurInfini

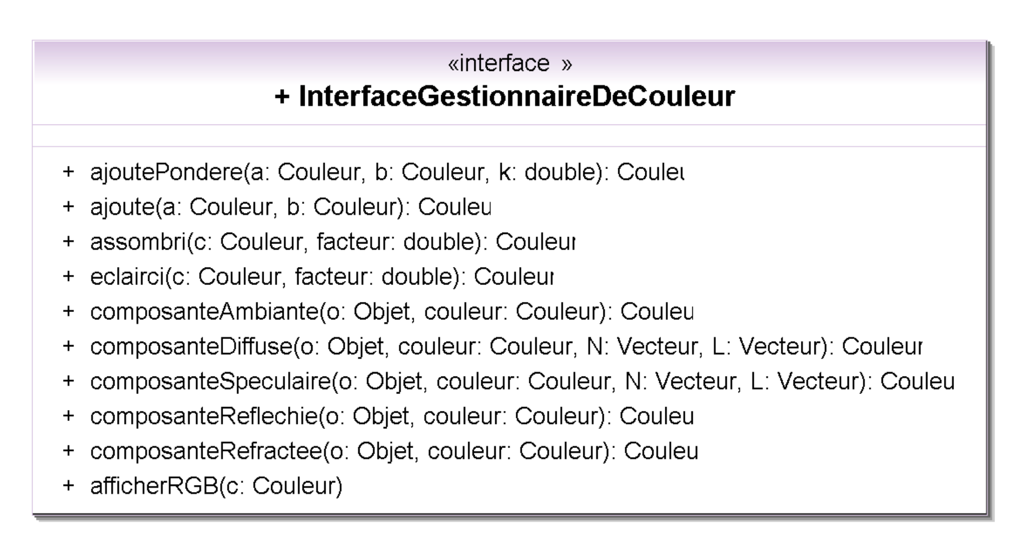


Fig. 4.7 – L'interface que doit satisfaire tout modèle de couleur

4.4.2 L'algorithme de détermination des couleurs

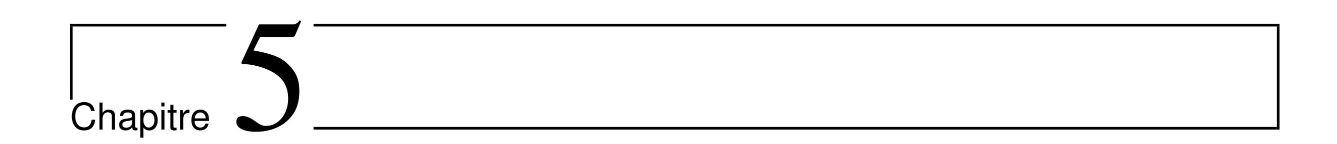
L'algorithme suivant part de la fonction *calculeCouleur* de la classe *Camera*.

- Depuis l'observateur(caméra observée), on émet des rayons R_i passant par chaque point de l'écran.
- On appelle la fonction $R_i.objetIntersecte()$, soit P_i le point d'intersection le plus proche s'il existe.
- En P_i
  - On calcule le rayon réfléchi et réfracté
  - On lance des rayons dans toutes les directions des lumières.
  - On rattrape ces rayons uniquement s'ils n'intersectent aucun objet entre la source et P_i, chaque objet de la scène étant considéré opaque. Dès lors on peut calculer via un *GestionnaireDeCouleur*, les différentes composantes de couleur dues aux sources de

lumière, en P_i

- Si les rayons réfléchi et réfracté intersectent un objet de la scène à partir de P_i (en tenant compte de la transparence des objets), on lance deux appels recursifs en ces points, et la couleur obtenue est alors ajoutée à la couleur due aux sources de lumière.

Toutefois, si à l'avenir cette fonction devient trop conséquente, nous créerons certainement une classe à part entière pour réaliser le calcul de lancer de rayon et donc le calcul de la couleur d'un point.



Importants diagrammes de séquence

Nous n'avons pas souhaité être exhaustifs et présenter dans cette partie tous les diagrammes de séquence dans tous les cas possibles. Nous avons cependant retenu deux cas bien particuliers qui ont lieu presque à coup sûr à chaque utilisation : Le chargement d'un fichier et la mise à jour d'une observation (à l'aide du bouton Rafraîchir). Ces deux diagrammes nous ont parus intéressants car ils solicitent les différents composants du programme.

5.1 Le chargement d'un fichier

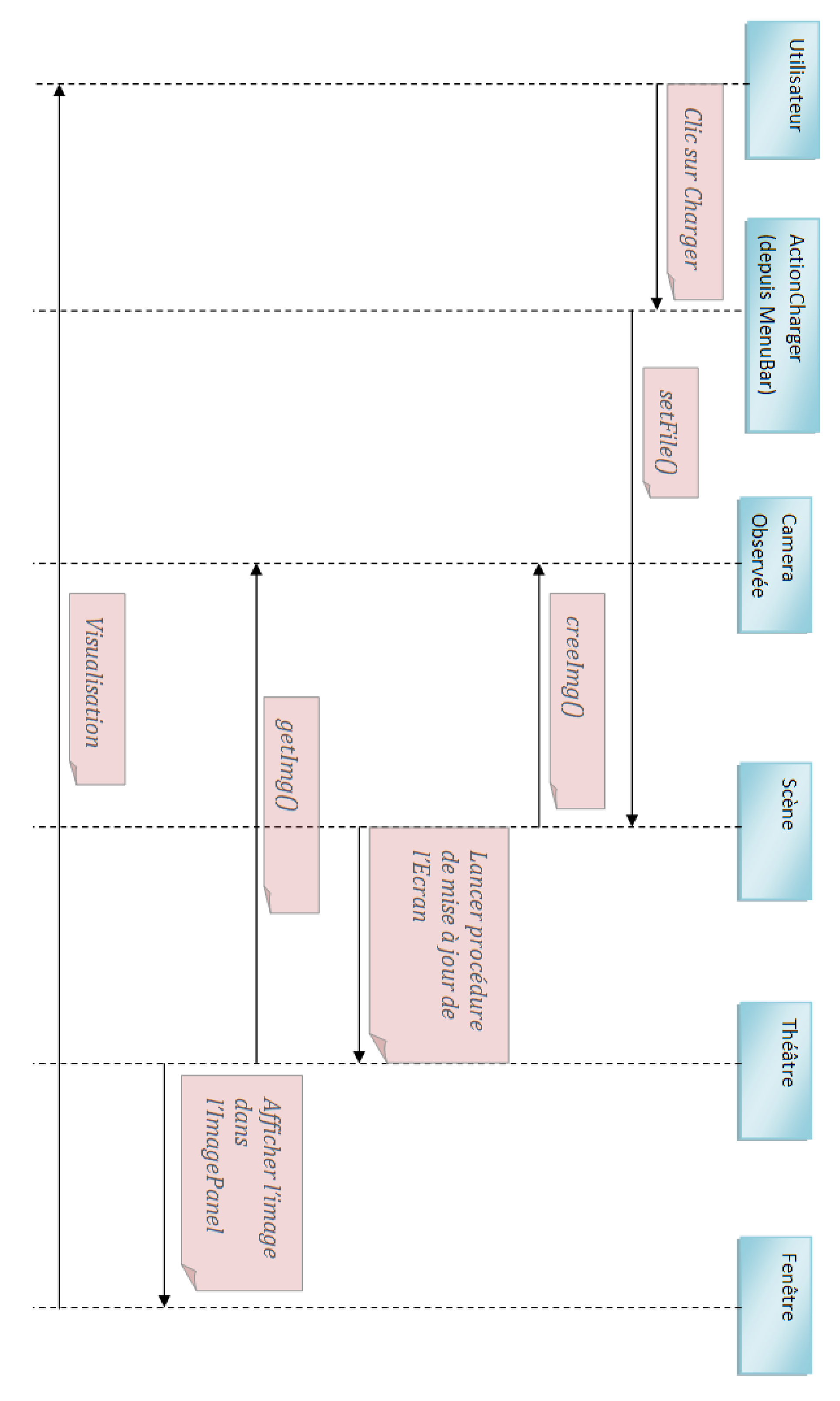


FIG. 5.1 – Diagramme de séquence de l'ouverture d'un fichier

5.2 La mise à jour du théâtre

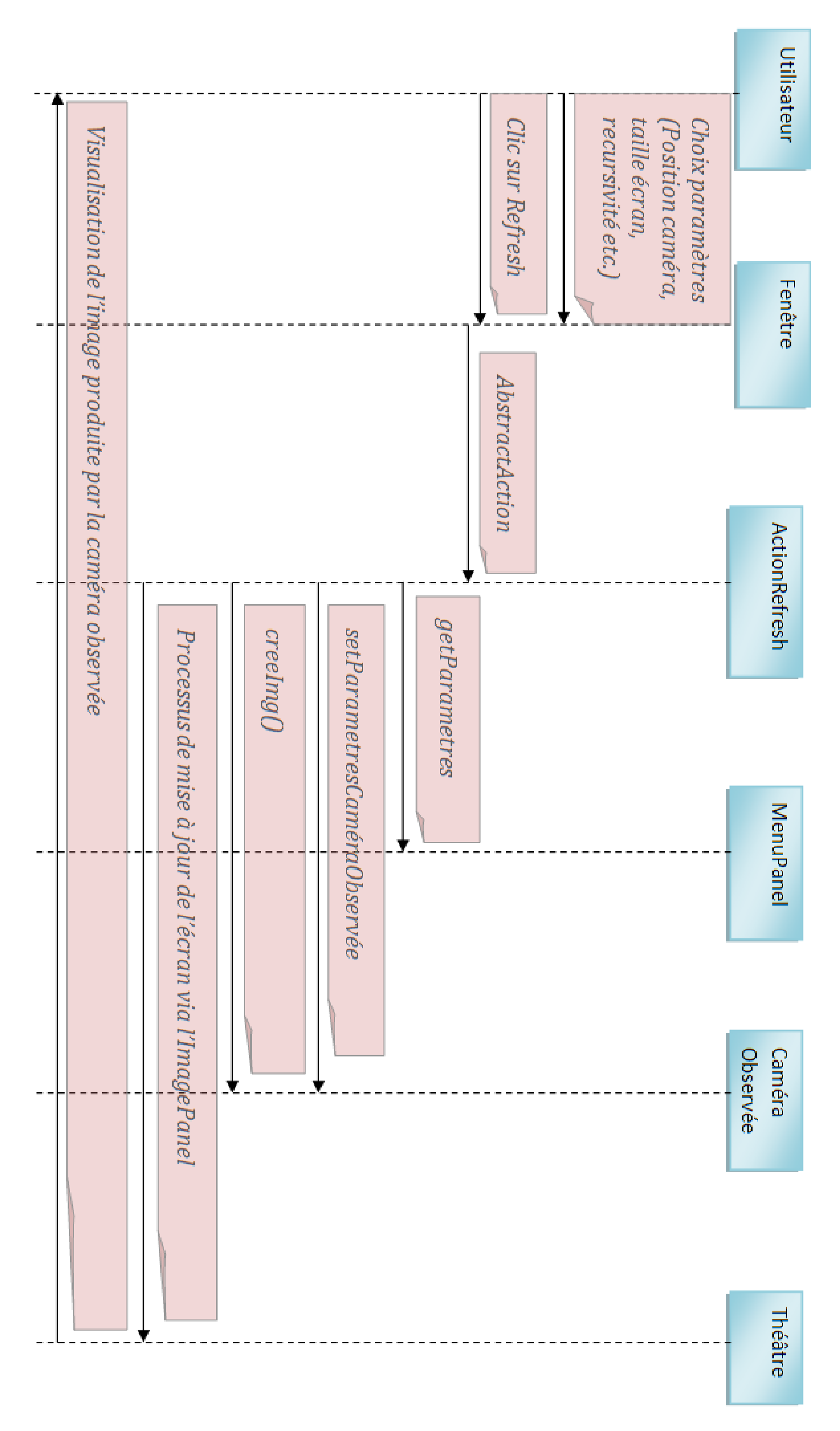
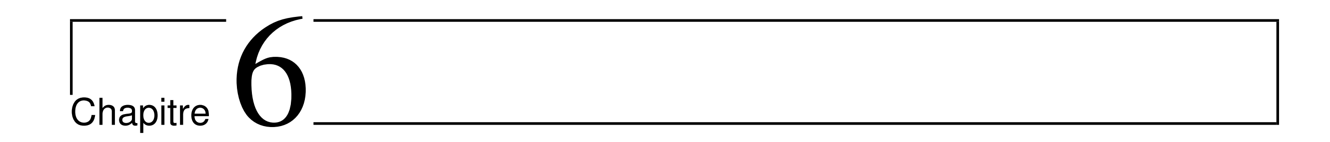


Fig. 5.2 – Diagramme de séquence de la mise à jour des paramètres



Le plan de dévelopement

Pour mener à terme le plan de dévelopement d'une application correcte, il va nous falloir continuer à raisonner par paquetages. Evidemment, la subdivision modérée de ces paquetages sera un plus d'une part dans l'écriture du code, mais également dans les retouches qu'on peut désirer y faire. Le code sera plus lisible et plus accessible à celui qui voudra s'y plonger. On pourra remarquer qu'au sein d'un package, l'attitude que l'on adoptera systématiquement sera de d'abord developper les classes principales, classes abstraites, ou interfaces. Puis, une fois avoir pris conscience des reéls besoins des "superclasses", on pourra passer au developpement des sous-classes sans problèmes, car les besoins seront ciblés.

6.1 Les calculs

Par calculs, nous sous entendons les méthodes principales qui interviendront dans le lancer de rayon. C'est à dire la méthode de calcul de l'objet intersecté par un rayon : il faut choisir l'objet intersecté le plus proche, prendre en compte le fait qu'un rayon qui part d'un objet peut être amener à intersecté ce même objet. On entend également la méthode de calcul de couleur, qui est récursive. Au maximum, nous essayerons de décomposer ces méthodes en sous méthodes élémentaires, pour une plus grande facilité de tests et lisibilité. Toutefois, pour vraiment tester ces fonctions, on aura besoin des objets, et si possible de l'interface graphique. Donc le developpement de ce coeur ne peut pas intervenir dès le début. Il nous faudra consacrer un temps précieux à décrire les algorithmes de manière très précise afin de préciser les méthodes sur les objets et les rayons dont on aura besoin.

6.2 Le théâtre

Le package théâtre, sera celui a developper en premier. Rappelons que dans celui-ci, on trouve la classe *Theatre* mais également dans les sous packages : *Scene*, *Camera*, *Lumiere*, *Objet*, *Sphere*, etc. Après le developpement du *Main* (qui est très sobre, comme dans l'avons vu en 1.1.1), le devloppement de la classe *Theatre*, qui met au point l'initialisation d'une scène et des caméras, les besoins que l'on auraient pu omettre seront alors révélés. Nous pourrons passer au developpement de *Camera*, *Scene* puis des objets. Cela permettra de les tester rapidement tout en les incluant dans la logique des classes dont ils sont les composants essentiles. Une des choses à surveiller sera les mises à jour. Si je modifie ma scène, il faut que je fasse attention à bien modifier mon theatre du même coup et inversement. Même si ces classes peuvent s'implémenter dans différents fichiers, il faut absolument penser à toutes les associations, aux héritages et aux dépendances qui les incombent.

6.3 L'interface graphique

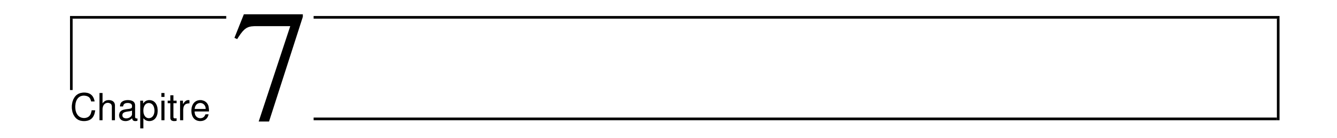
Pour l'interface graphique, a ce jour nous avons prévu d'insérer : le chargement du fichier des caméras, le chargement du fichier de la scène, des champs pour modifier la position de l'observateur, pour modifier la direction de son regard, pour modifier la position de l'écran et sa résolution, un champ pour modifier la profondeur du lancer de rayon et un bouton de mise à jour.

Nous avons par ailleurs décidé de garder une taille d'image affichée à l'écran fixe. Ainsi, même si l'image à une résolution inférieure à celle de l'écran, l'image sera agrandie pour s'ajuster à la taille de l'écran.

Le developpement de l'interface graphique peut se faire indépendemment, dans un premier temps, du developpement du théâtre. Leur principaux moyens d'échanges ont été révelé dans les diagrammes de séquences. Nous ecrirons d'abord le code de la classe $Fenetre$, en mettant des $JPanels$ vides, où on le souhaite. Puis, nous passerons au developpement des $JPanels$ en questions.

6.4 Le lien entre les deux

L'ultime phase sera celle où l'on créera les liens entre interface graphique et théatre. Le théatre à besoin de connaitre l'écran ($ImagePanel$), dans lequel il va afficher l'image. L'interface graphique à besoin d'accéder aux paramètres de la scène et des caméras afin de mettre à jour ces paramètres via des *ActionListener* qui seront alors developpés.



Le Plan de test

Nous allons procéder aux tests des différentes classes dans l'ordre qui a été décrit plus haut. On peut distinguer deux phases dans le déroulement de ces tests : - La phase de test unitaire (chaque classe est testé individuellement) - La phase de test d'intégration (test du logiciel dans sa globalité)

7.1 Les tests unitaires

Pour tester les classes du package objet, nous allons créer ces objets (sphères par exemple), vérifier les règles d'intersection, réflexion et réfraction.

Pour tester notre classe vecteur, nous allons créer plusieurs d'entre eux, et effectuer des produits scalaires, des translations etc. nous testerons nos méthodes et vérifierons les résultats par des calculs à la main.

Ainsi de suite, pour chacune des classes élémentaires. Ce seront générallement des tests simples, mais permettant de réveler tout de suite des erreurs qui pourraient être d'une grande conséquence par la suite, et difficilement décelables.

7.2 Les tests d'intégration

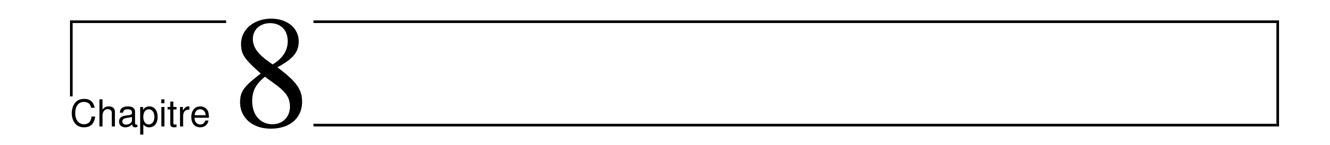
Il va falloir vérifier le bon fonctionnement de nos classes et méthodes, mais surtout le bon fonctionnement des intéractions entre classes. Nous allons pour cela devoir charger des fichiers de scène et de caméra différents. L'avantage est qu'une fois qu'on affiche la scène selon un certain point de vue, et qu'on la définit à partir de ce point de vue, on se rend vite compte des erreurs commises lorsque par exemple on tourne de 90 ou 180 degrés autour de cette scène. L'aspect pratique de cette réalité géométrique facilite grandement le repérage d'erreurs. Ceci nous permettra de tester entre autres la classe camera.

Concernant l'interface graphique, il nous suffit de vérifier le bon fonctionnement et la bonne prise en compte du "clic" sur bouton.

Il faudra ensuite passer aux étapes de vérification de la saisie de données de l'utilisateur. Rentre-t-il bien une scène ? Donne-t-il bien une caméra ? etc.

Troisième partie

Le programme final et ses évolutions



Chapitre 8

Manuel d'utilisation du programme

8.1 L'interface

D'une manière générale, voici l'écran que l'on obtient lorsque on a chargé une scène et des caméras à l'aide de fichiers externes.

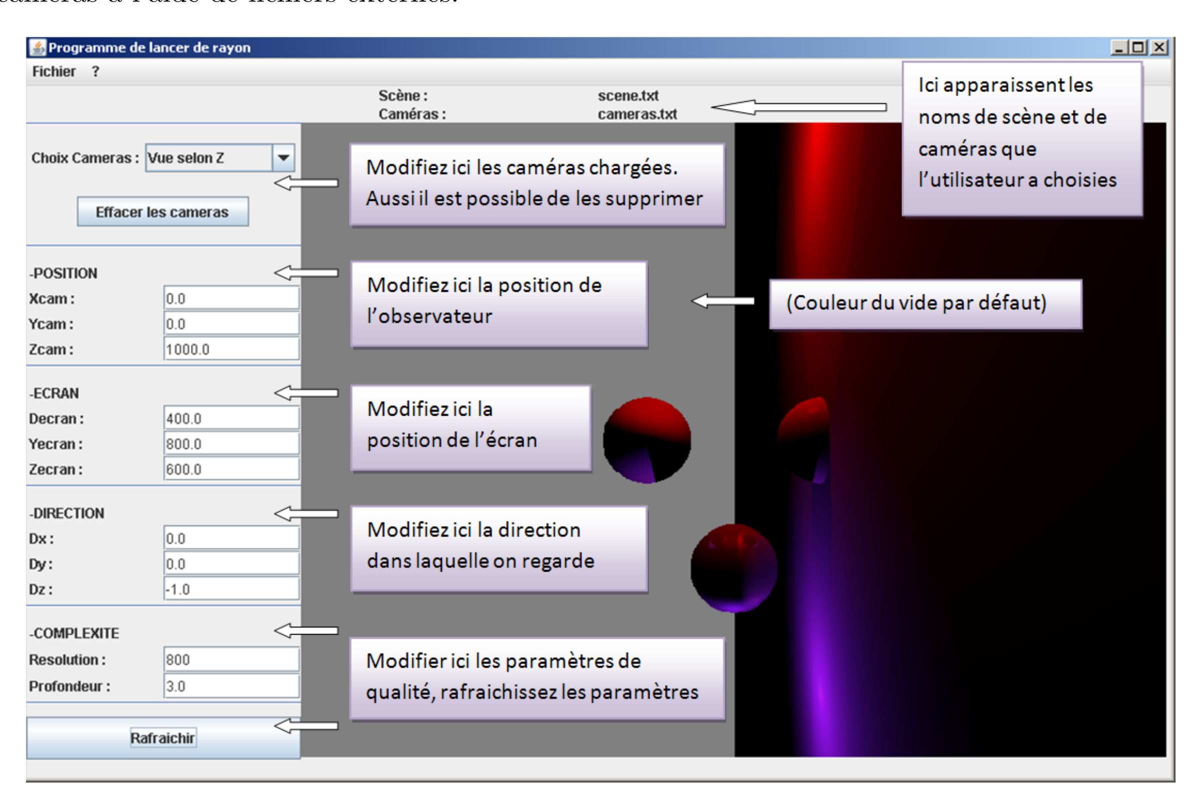


Fig. 8.1 – Champs de l'interface d'utilisation

La plupart des paramètres sont réglables directement depuis l'interface où l'on observe la scène reconstituée. La position de l'observateur est sa position dans l'espace (x,y,z) dans le même repère que la scène et les objets qui la compose. Ce que nous appelons ici *Observateur*, est le point source de la *Camera* observée. La direction d'observation correspond au vecteur de l'espace selon lequel on regarde. Les paramètres de l'écran sont la largeur $Yecran$, sa hauteur $Zecran$ et la distance à laquelle il est de l'observateur $Decran$. Etant donné qu'on peut charger plusieurs caméras depuis un fichier, "le menu" choix caméras permet d'opter pour une seule caméra à la fois.

Considérons maintenant les deux menus qu'on peut trouver dans le coin supérieur gauche de l'interface.

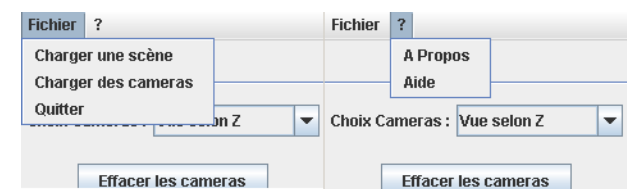


FIG. 8.2 – Menu fichier et ?

Dans le premier (menu "Fichier") on peut charger les caméras et la scène, il suffit de rentrer le nom de fichier correspondant (attention à l'extension *.txt*) qu'on laisse le soin à l'utilisateur de créer si les modèles fournis ne conviennent pas. Dans le second (menu "?"), deux rubriques sont proposées, mais sans réelle pertinence à ce jour si ce n'est le nom des créateurs du logiciel.

8.2 La structure des fichiers chargés

Pour chacun des fichiers, nous indiquerons dans un tableau les paramètres qui doivent être entrés, et nous donnerons également un exemple de fichier. Nous rappelons que chacun des paramètres sont séparés par des tabulations.

8.2.1 Le fichier contenant les caméras

Xobs	Yobs	Zobs	Yecr	Zecr	Decr	Resolution	Dx	Dy	Dz	Recursion
dbl	dbl	dbl	dbl	dbl	dbl	int	dbl	dbl	dbl	int

```
%%% camera
Vue selon X
1000  0 0 800 600 400 400 -1  0 0 3

%%% camera
Vue selon moinsX
-1000 0 0 800 600 400 400 1 0 0 3
```

8.2.2 Le fichier de scène

Pour une sphère :

x	y	z	rayon	Kreflection	Ktransparence	R	V	B	n	Reflect	brillance
dbl	dbl	dbl	dbl	dbl	dbl	char	char	char	dbl	dbl	char

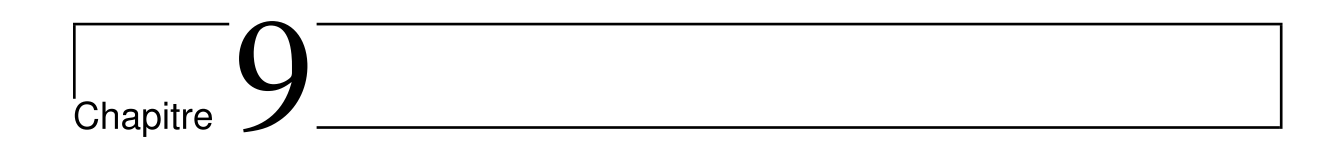
Pour un plan ou une facette, on a d'abord les trois coordonnées de trois points sur les trois premieres lignes, puis les paramètres suivants sur une quatrième ligne.

Kreflection	Ktransparence	R	V	B	n	Reflect	brillance
dbl	dbl	char	char	char	dbl	dbl	char

```
%%% Sphere
300 0 0 100 0.1 0.2 122 255 255 1.2 0.8 128

%%% Plan
```

```
-300  0 0
-300  1 0
-300  0 1
1 0 255 255 255 1 0.5 128
```



Le fonctionnement en profondeur du programme

Dans ce chapitre, nous allons tâcher d'expliquer le fonctionnement de notre programme, de la façon la plus claire possible. Pour ce faire nous emploierons principalement des schémas canoniques : diagrammes de séquences de sous cas d'utilisation usuels, et diagrammes de classes présentant les méthodes, attributs et relations entre différentes classes. Veuillez noter que sur ces schémas n'apparaitront pas toutes les méthodes. Typiquement les $getAttribut()$ et $setAttribut()$, qui peuvent s'avérer nombreux, et dont la présence allourdit inutilement les diagrammes ne seront pas représentés. De même nous avons supprimé les paramètres qui n'apportent rien, ou qui allourdissent la classe. C'est le cas de attributs *Swing*, mais également des intermédiaires de calcul que l'on stocke sous forme d'attribut de méthodes pour pouvoir les utiliser dans toute la classe : $tailleEcran$, $nbCameras$, $vecteurNormal$ etc. Par ailleurs, nous décrirons les algorithmes principaux du programme, afin que le lecteur puisse comprendre comment nous avons choisi de gérer le lancer de rayon. Nous ne tenons pas à rentrer dans des considérations trop complexes et exhaustives, et nous nous contenterons d'appuyer nos explications sur des cas pretinents. Le lecteur pourra alors comprendre l'esprit global du programme. Ainsi par exemple, le chargement de fichier de scène ce fera d'une manière similaire au chargement du fichier de caméra. Nous tenons également à rappeler que le schéma 4.1 du chapitre 4.1 permet une vision globale du programme en remarquant les deux composantes parallèles constituées d'une part de l'interface graphique, et du theatre d'autre part.

9.1 Algorithmes canoniques

9.1.1 Calcul de l'objet intersecté le plus proche

Ce calcul est effectué au sein de la classe *Rayon*. etant donné une origine I, et une direction $\vec{d}$, il s'agit de parcourir le vecteur $objets(Vector < Objet >)$, et de regarder si l'objet est intersecté, via sa méthode $estIntersecte()$. Le problème réside toutefois dans le traitement du cas où le rayon part d'un point qui est à la surface d'un objet. Dès lors, l'intersection la plus proche serait le point lui même, ou un très proche voisin dû aux imprecisions numériques. Pour contourner cela, on effectue un test. Si le rayon que l'on considère contient déja un $ObjetIntersecte$, on ne tient pas compte de cet objet dans le parcourt du vecteur $objets$. Avant l'appel de $rayon.calculObjetIntersecte()$, on aura d'abord effectué un $rayon.setObjetIntersecte(ObjetOuJeSuis)$, pour préciser que l'on part d'un point qui est sur l'objet $objetOuJeSuis$. Problème qui en resulte : dans ce qui précède on ne prend pas en compte le cas où l'objet d'où je pars fait écran. Il faut faire des tests différents suivant que je sois sur

un objet de type Contenant, et selon que je rentre ou que je sorte de l'objet. D'où l'algorithme suivant :

```
#CLASSE RAYON
#METHODE calculObjetIntersecte(Vector<Objet> objets)
RETOURNE l'Objet INTERSECTE LE PLUS PROCHE
    SOIT I un Point;
    SI LE RAYON PART D'UN OBJET, CES DEUX PARAMETRES SONT NON NULL :
    SOIT objetPropre=objetIntersecte
    SOIT IPropre=Intersection;

    POUR(i=0;i<objets.size();i++)
      SI JE PARS D'UN OBJET, JE NE FAIS RIEN
      SINON
        I=objets.get(i).estIntersecte(this);
        SI(I!=null)
          ON STOCKE LE MINIMUM DE LA NORME ENTRE I ET L'ORIGINE DU RAYON, AINSI QUE L'OBJET INTERSECTE.
          A LA FIN DE LA BOUCLE FOR ON AURA :
          objetIntersecte=OBJET INTERSECTE LE PLUS PROCHE;
        FIN SI
      FIN SI
    FIN POUR

    SI(IPropre!=null) ie LE RAYON PART D'UN OBJET
      SI(
        (objetPropre.estUnContenant() ET objetPropre.rentreDansObjet(this))
        OU
        (rayon_precedent!=null ET !objetPropre.estUnContenant() ET JE PASSE A TRAVERS L'OBJET)
        )
      ALORS L'objetPropre FAIT ECRAN
            norme_min=0;
            Intersection=IPropre;
            objetIntersecte=objetPropre;
      FIN SI l'objetPropre fait écran
      SINON
            JE N'AI INTERSECTE AUCUN AUTRE OBJET DEPUIS L'OBJET SUR LEQUEL JE SUIS
            objetIntersecte=null;
            Intersection=null;
      FIN SINON
    FIN SI LE RAYON PART D'UN OBJET

    SI (objetIntersecte!=null)
      JE CALCULE LE RAYON REFLECHI ET REFRACTE
    FIN SI

    RETURN objetIntersecte;
```

9.1.2 Calcul de la couleur

Nous invitons le lecteur à consulter le schéma 9.1 de la section suivante pour suivre cet algorithme. On pourra observer que ce calcul a été décomposé en trois sous fonctions élémentaires qui sont décrites ci-dessous.

```
#CLASSE Camera

SOIT O un Point, la position de l'observateur
SOIT Y un Vecteur de base de l'écran
SOIT Z un Vecteur de base de l'écran

#METHODE rempliImage()
  SOIT PCourant un Point, initialement le coin supérieur
     gauche de l'écran
  POUR(i=0;i<limiteY;i++)
    POUR(j=0;j<limiteZ;j++)
      SOIT R le rayon O->Pcourant
      image.setRGB(i,j,calculeCouleur(R,0))
      Pcourant.translater(Z)
    FIN POUR
    Pcourant.translater(Y)
  FIN POUR
FIN METHODE

#METHODE calculeCouleur (R:Rayon, k:entier)
  SI k==profondeur RETURN NULL
  SINON
    SOIT Objet=R.calculObjetIntersecte(VECTEUR D'OBJETS);
    SI Objet==NULL RETURN new CouleurInfini()
    SINON
      RETURN calculeCouleurDueAuxLumieres(R,Objet,k)
    FIN SINON
  FIN SINON
FIN METHODE

#METHODE calculeCouleurDueAuxLumieres(R:Rayon, Oujesuis:Objet
   ,k:entier)
  SOIT P=R.getIntersection() intersection entre R et Oujesuis
  SOIT couleur une Couleur
  POUR(i=0;i<NLumieres;i++)

    SOIT Slum le Point de la Lumiere i
    SOIT Rlum le Rayon P->Slum
    Rlum.setObjetIntersecte(R.getObjetIntersecte());
    Rlum.setIntersection(R.getIntersection());
    Rlum.setRayonPrecedent(R);

    SOIT Objet O = Rlum.calculObjetIntersecte(VECTEUR D'
       OBJETS);

    SI Le Point est dans l'ombre d'un autre objet
```

```
      On assombri couleur
    SINON
      On calcule les différentes composantes de couleurs et
         on les ajoute à couleur

    FIN SI
  FIN POUR

  On lance si besoin est les rayons reflechis et refractés
  couleur_reflechi=calculeCouleur(Rreflechi,k+1);
  couleur_refracté=calculeCouleur(Rrefracté,k+1);
  On ajoute ces couleurs à couleur

  RETURN couleur
```

9.2 Un dessin vaut mieux qu'un long discours

9.2.1 Le calcul de l'image de la caméra observée

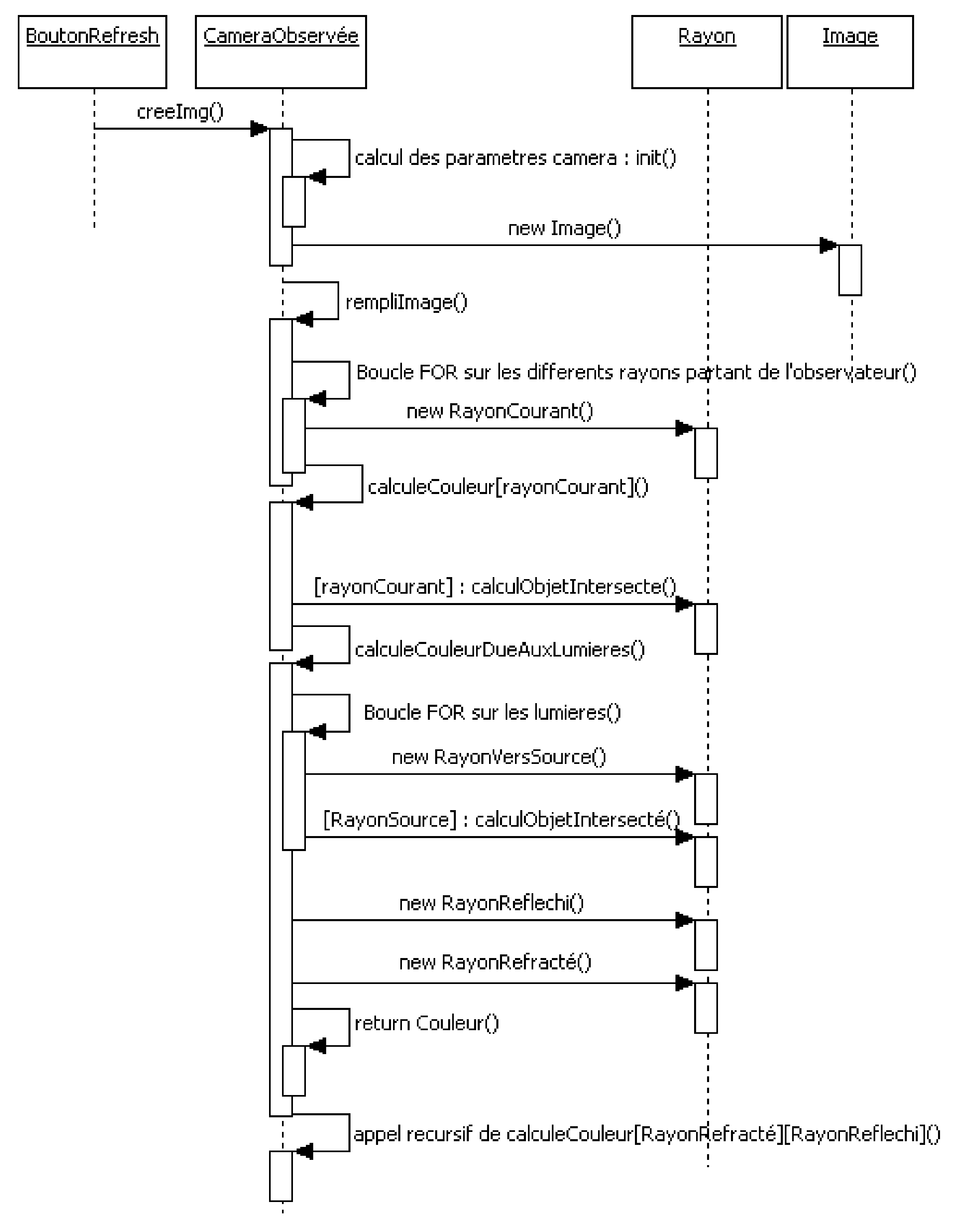


Fig. 9.1 – Le calcul de l'image de la caméra observée

9.2.2 Le chargement des caméras et les exceptions

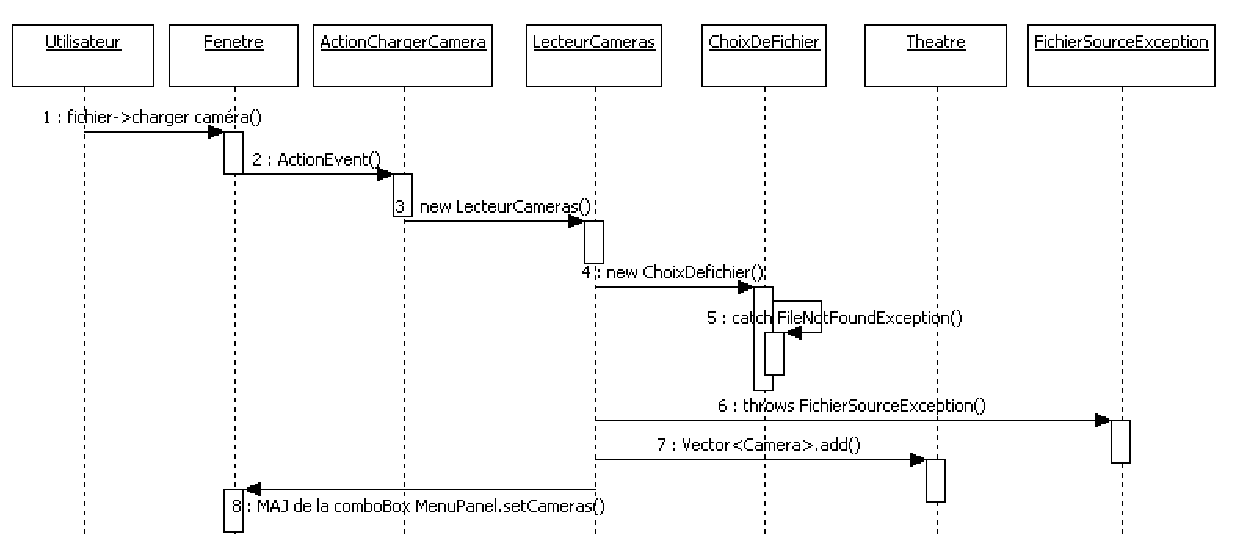


FIG. 9.2 – Diagramme de séquence du chargement des caméras

9.2.3 Diagramme de classe de l'interface graphique

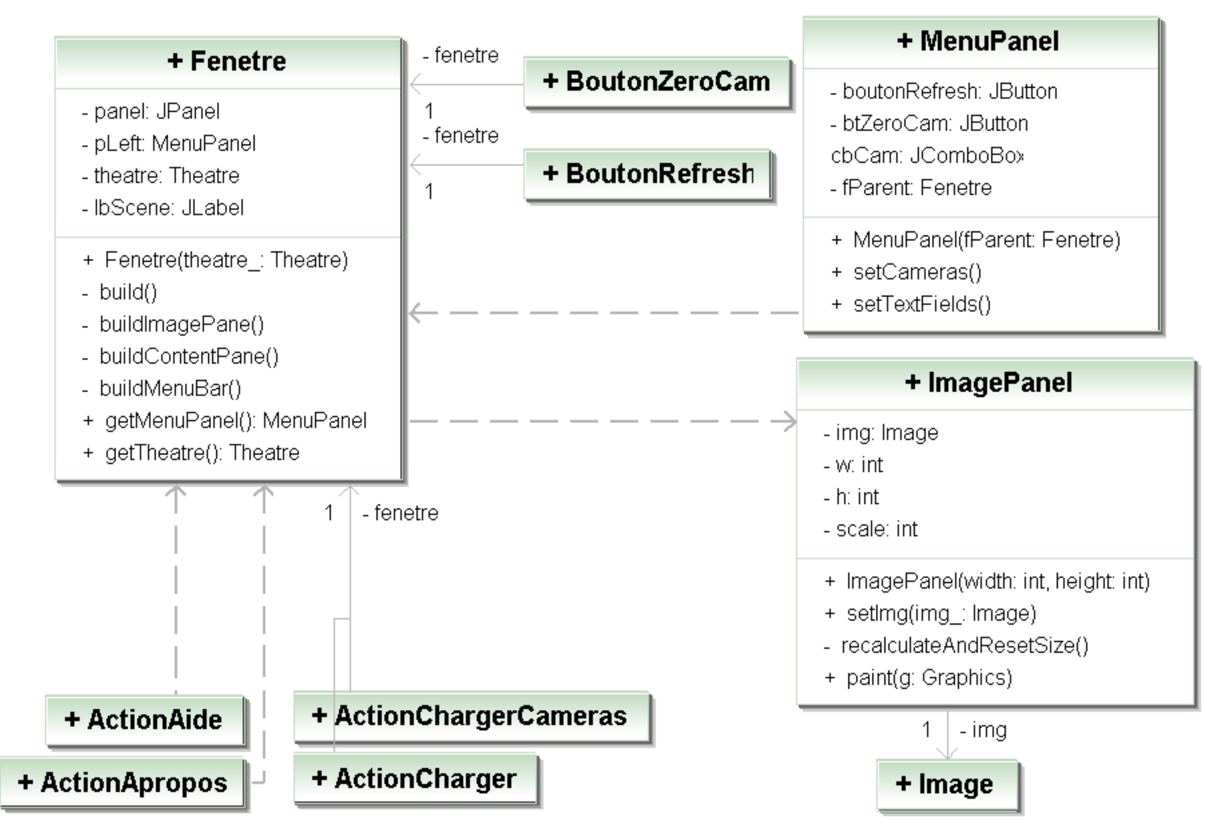


FIG. 9.3 – Diagramme de classe de l'interface graphique

9.2.4 Diagramme de classe du theatre

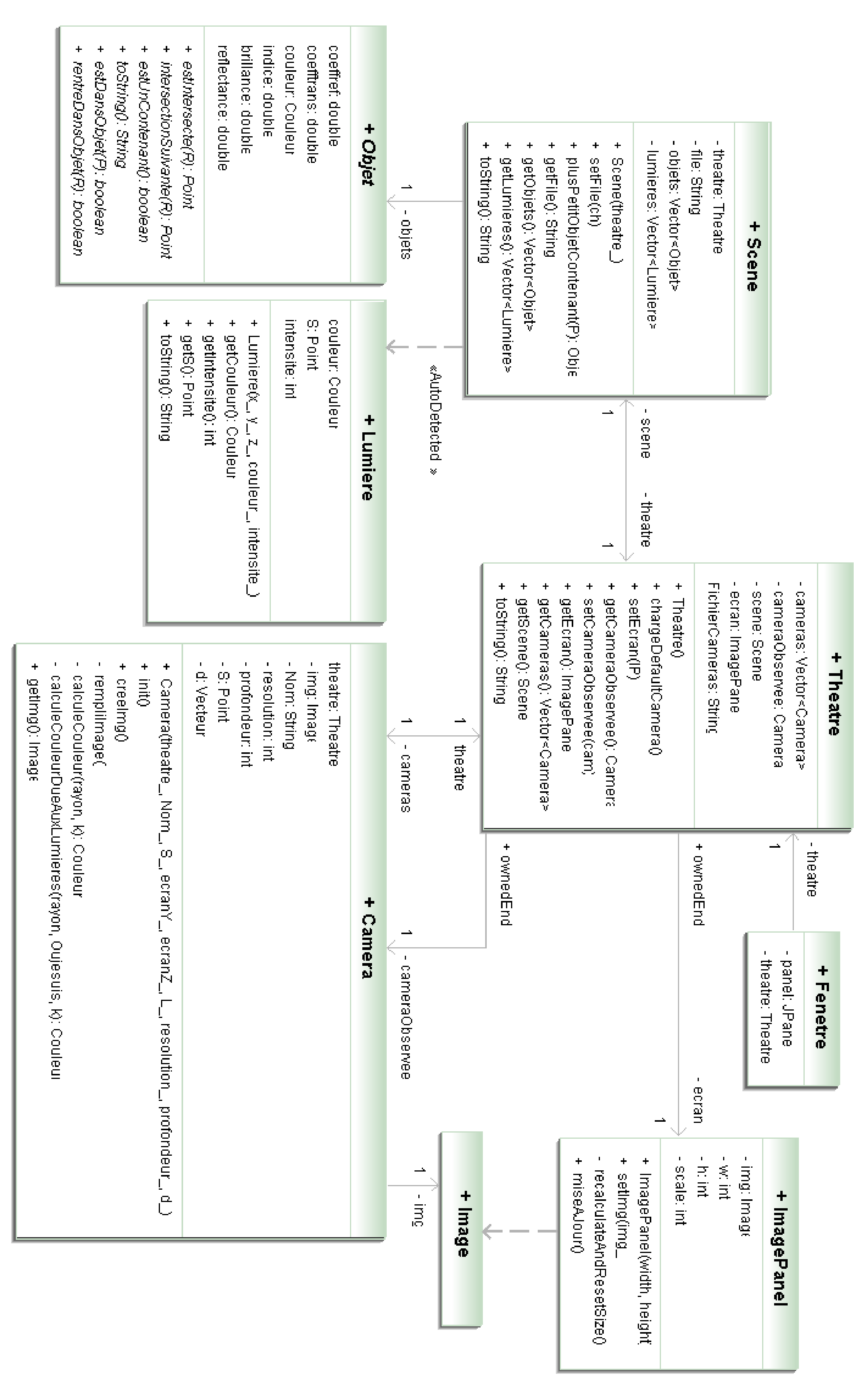
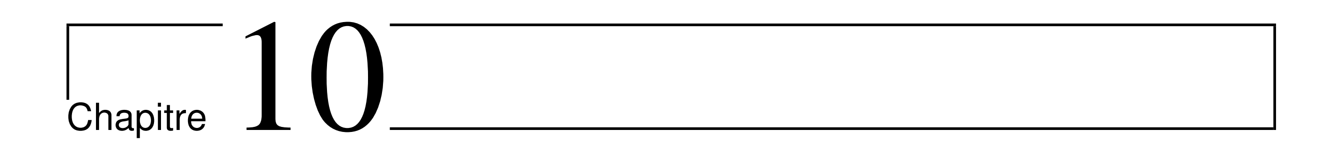


Fig. 9.4 – Diagramme de classe du theatre



Chapitre 10

Les evolutions entre la phase de conception et l'implémentation

Il n'y a pas eu de réels changements entre ces deux phases comme pourra constater le lecteur en comparant le contenu des classes de la partie deux de ce rapport avec le rapport de conception donné deux mois auparavant.

10.1 Ajout d'un nouveau package

Nous avons choisi de dégager le fait que nous avions gérer les exceptions, en créant deux classes chargées d'avertir l'utilisateur que la structure des fichiers sources est incorrect, que le format de celui est incorrect, ou que les paramètres rentrés dans l'interface graphique ont un mauvais format.

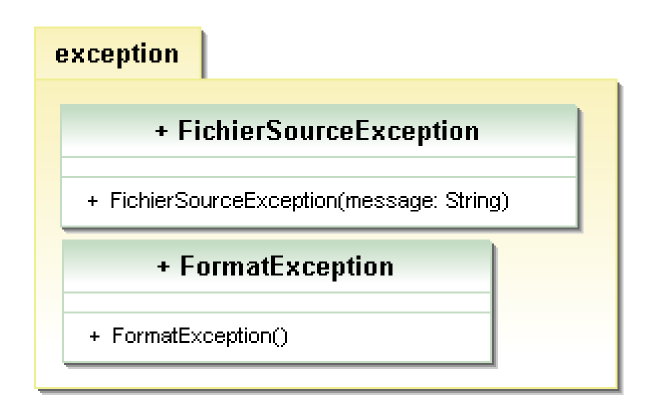


Fig. 10.1 – Les classes du packages exceptions

10.2 Suppression de l'écoute clavier

Petit rêve d'enfant, nous pensions qu'il serait possible de permettre à l'utilisateur de se déplacer dans le décor en utilisant les fleches du clavier et du pavé numérique. Devant la longueur du temps d'execution, nous avons du renoncer à cela.

Conclusion

A l'issue du rapport d'analyse, nous avions défini l'ensemble des scénarios d'évolution du programme pour un cas d'utilisation normal. Après le rapport de conception, nous avions avancé ce travail dans la définition de nos classes principales et précisé les interactions jusqu'au niveau des attributs et méthodes. Il nous apparaît clairement que cette étape est indispensable avant le codage proprement dit puisque nous nous donnons la possibilité de répartir efficacement le travail via les définitions de tous les noms d'attributs et méthodes. Ce travail préalable fut un outil de développement d'une grande utilité, et nous avons pu aborder sereinement les phase de codage et de tests. Nous sommes satisfaits car notre programme fonctionne, et les phases de test ont été effectuée progressivement et en harmonie avec les rapports précédents. Par ailleurs, ce travail fut enrichissant de part les connaissances et l'expérience qu'il nous a apporté. Celui-ci fut même interressant pour nous deux de part son côté graphique et le sens physique associé à la modélisation de la lumière.